
Higher-Order and Symbolic Computation, 13, 355–368, 2000

Glossary for Partial Evaluation and Related Topics

TORBEN Æ. MOGENSEN (ED.) torbenm@diku.dk
DIKU, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen East, Denmark

Abstract. Most areas of research or work use their own set of words and phrases and gives specific technical meaning to terms that in everyday speech may mean something less specific or something else entirely. The area of partial evaluation and program transformation is no different, which may make it hard for the uninitiated to grasp some parts of technical papers or discussions. This list of words and terms is intended as a help to people new in the subject area, but may also be of help to experienced researchers, as different research groups tend to develop different terminology.

Keywords: partial evaluation, program analysis, program specialization, program transformation

Introduction

The following is an alphabetical list of common terms and phrases concerning partial evaluation, program transformation and related topics. It is based on the terminology list composed by Torben Mogensen and Carsten Kehler Holst at the first partial-evaluation workshop in 1987 [1]. Numerous new words and phrases have been added and a few terms no longer used have been deleted. Several entries have been clarified or modified to reflect current usage.

When a listed term is used exclusively in the literature about a specific language or class of languages, the explanation of the term may use language-specific terminology without further explanation.

The list is mainly intended as an aid when reading literature about partial evaluation and related subjects. As such, it is thus more descriptive than prescriptive. The list is not intended either to replace an annotated bibliography, so it contains no bibliographic references.

An "(obs.)" marking after an entry means that the term is not widely used anymore and should in most cases be replaced by a more current term.

Contributors

Thanks are due to the following people for their contributions and clarifications: Elvira Albert, Maria Alpuente, Olivier Danvy, John Gallagher, Robert Glück, Bernd Grobauer, Neil Jones, Luke Hornof, Julia Lawall, Michael Leuschel, Dave Sands, Walid Taha and Peter Thiemann.

List of terms

Abstraction: Sometimes used as a synonym for *generalization*.

Abstract interpretation: A technique for the static analysis of programs. Intuitively it can be thought of as "abstractly executing" a program on abstract values, in order to

obtain a safe approximation of the program's concrete behaviour. More precisely, it is an approximation of some semantic model of the program's behaviours, and is typically computed by the construction or approximation of the fixed points of some system. Abstract interpretation has been used to perform *binding-time analysis*.

Abstraction-based program specialization: The relationship between *abstract interpretation* and program *specialization* has been observed and formal frameworks supporting this idea have been developed. Abstraction-based program specialization combines these two approaches and can thereby obtain specialization and *analysis* which are outside the reach of either method alone.

Action: An *annotation* that is in the form of an explicit indication of how the annotated term should be treated during *partial evaluation* or similar transformations.

Annotation: Generally, any structured data that is attached to some part of a program text. For example, type inference commonly results in "type annotations" for the program's components, or a C programmer may supply a compiler with hand-made annotations about function inlining. When the word "annotation" is used without qualification in *partial-evaluation* contexts, it usually means annotations that direct the choice between several possible ways for the *specializer* to handle a program construct. For example, they could concern the *binding times* of variables and expressions, or the *unfoldability* of function calls. The annotations can be supplied manually or created by automatic analyses.

Arity raising: Applying *structure splitting* to parameters of functions, hence (potentially) increasing the arity of the functions.

Autoprojector: (obs.) A synonym for *self-applicable* partial evaluator.

Bifurcation: (obs.) A synonym for *binding-time separation* (the second listed meaning).

Binding time: The time at which a variable, expression or parameter gets a concrete value. When talking about compilers, binding times are compile time, link time and run time. In terms of *partial evaluation*, variables bound at the time partial evaluation is performed are called *static* whereas variables bound when the residual program is executed are called *dynamic*. See also *stage of computation*.

Binding-time analysis: An analysis that makes decisions about the *binding times* of variables, expressions etc. in a program before the concrete values with respect to which it is specialized are known. Binding-time analysis is used in an *offline partial evaluator* and is usually based on an approximation of the binding times that would occur in an *online partial evaluator*.

Binding-time improvement: A transformation applied to a program with the purpose of performing parts of the computation at an earlier *binding time*. Instances of binding-time improvements are *eta-expansion* and *The Trick*.

Binding-time pattern: A set of binding-time annotations that describe the binding times of all components of a single object, e.g., all the parameters to a function or all variables at a *program point*.

Binding-time separation:

1. The degree to which *static* and *dynamic* computations are separated in a program, i.e., how well they are *staged*. A high degree of binding-time separation is typically required for successful *specialization*.

2. An implementation technique for *partially static data*. It involves a *preprocessing* of *binding-time annotated* programs to replace each partially-static variable with a purely static variable holding the static part and a purely dynamic variable holding the dynamic parts. This allows the use of a *specializer* that has no explicit support for partially static data.

Binding-time signature: The *binding-time annotated* type (signature) of a function.

Binding-time types: *Binding-time annotations* in the form of non-standard type systems. In this context, *binding-time analysis* is often done by a form of type inference.

BTA: Abbreviation for *binding-time analysis*.

Cache: See *data specialization*.

Characteristic tree: Used in *partial deduction* to refer to an abstraction of a (partial) SLDNF-tree which only remembers the position of the selected literal and the clauses that have been resolved. Used for controlling the *generalization* process: instead of examining the syntax of *configurations*, one uses the computational behaviour of configurations. If two configurations have a different behaviour, generalization should be avoided. On the other hand, if two configurations have the same behaviour one attempts to construct a single generalization with the same behaviour.

Closedness: A condition needed to ensure the strong *soundness* and *completeness* of *partial evaluation* of (functional) logic programs. It guarantees that all calls which might occur during the execution of the specialized program are covered by some program rule. In *functional logic programs*, the standard, LP-like closedness condition which simply checks that the whole expression is an instance of one of the specialized calls, is generalized by recursively inspecting all subterms of the considered expression. Closedness is essentially equivalent to the "need for folding" condition of *fold-unfold* frameworks.

Cocom: (obs.) See *cogen*.

Code blowup/code explosion: The phenomenon that a residual program can be substantially larger than the original program. Often caused by excessive *unfolding* or creation of *variants*. See *polyvariant*.

Code sharing: Reusing a previously-generated specialized *program point* instead of *specializing* the *program point* anew. With term-rewriting transformation techniques such as *supercompilation*, the word *folding* is preferred instead.

Cogen: Short for "Compiler Generator". Often used for a generator of *generating extensions*, since it can be used for generating compilers from interpreters. Alternative abbreviations for these generators of program generators include *cocom* (compiler-compiler), *gegen* (generating-extension generator), and *pecom* (partial-evaluation compiler).

Compile-time specialization: Sometimes used to refer to specialization as a source-to-source process (such that the residual program must be compiled before it can run), as opposed to *run-time code generation*.

Completeness: The *residual program* is able to compute, for the intended queries, all answers (and values) which can be produced by the original program. An essential ingredient for completeness is the compositionality of the semantics, i.e., *partial evaluation* of (*functional*) *logic programs* is complete whenever the underlying *narrowing* strategy is compositional.

Configuration: A symbolic representation of a set of computation states or function calls. Mostly used in literature about *supercompilation*.

Congruence: (obs.) A condition on *binding-time annotations* that requires the *static* parts of a program state to be computable from the static parts of the previous states.

Conjunctive partial deduction: An improvement on *partial deduction* which avoids splitting the conjunctions at the leaves of a partial SLDNF-tree before the next *specialization* step, hence allowing specialization of entire conjunctions of calls instead of just individual calls. This enables conjunctive partial deduction to achieve both *deforestation* and *tupling*. Inspired by *supercompilation*.

Constant folding/constant propagation: Evaluating subexpressions that only depend on constants and replacing these by the obtained values. A common compiler optimization.

Constructor specialization: A variant of *partial evaluation* where constructors in a datatype are specialized to parts of their arguments. The result is a *residual* datatype that may contain more constructors than the original datatype. In the residual program, dynamic pattern matching on the original constructors is replaced by pattern matching on the specialized constructors.

Context sensitive: Used to describe analyses that process the same function call in several different calling contexts instead of combining these into a single most general context.

Continuation-based partial evaluation: A technique used in *partial evaluation* where continuations are used to move *static* computations across *dynamic* contexts (e.g., let expressions or case expressions).

Cross-stage persistence: The ability, in a later *stage of computation*, to use a value computed in one stage. This generalizes *lifting*, as the value need not change representation if the stage where it is used is part of the same execution process as the stage that used it. Hence, cross-stage persistence may apply to values that cannot in general be *reified*/*lifted*, such as pointers or functional values.

Currying: Changing a function that takes a pair of arguments to take two consecutive arguments, i.e., changing its type from $(\alpha \times \beta) \to \gamma$ to $\alpha \to \beta \to \gamma$. Currying differs from *partial evaluation* mainly because a curried function does not perform any computation until both arguments are provided. In a language with full laziness, *syntactic currying* can, however, achieve some of the effects of partial evaluation.

Data specialization: An instance of *mixed computation* that only performs data-flow optimizations. The first phase (the "loader") performs early computations and passes the results in a data structure (the "cache") to the second phase (the "reader"), which performs the remaining computations. The reader is independent of the data used in the data-flow analysis and can hence be reused with different data. This independence contrasts with *partial evaluation*, where each static data set yields a new *residual program*.

Deforestation: A technique for *fusion* of a simple class of functions operating over inductive data types. The name refers to the fact that intermediate trees are eliminated.

Division: (obs.) A description of the *binding times* of a set of variables. See also *initial division*.

Driving: Unfolding of nested function calls controlled by how the outer function uses its input. Driving is usually combined with *folding* and *generalization* to form *supercompilation*.

Dynamic: Having a late *binding time*.

Dynamize: A synonym for *residualize*. The latter word is preferred.

Eta-expansion: In terms of lambda calculus, replacing a term t by $\lambda x.t\ x$ where x does not occur free in t. Eta-expansion separates t from its context, and thus a binding-time analysis may assign different binding times to both. Eta-expansion can be generalized to other kinds of objects, e.g., pairs $t \rightarrow$ (first t, second t) and sums $t \rightarrow$ case t of inl $x \rightarrow$ inl x | inr $y \rightarrow$ inr y.

Extension: Properties of an object that determine how it may be applied or used, independent of its actual representation. For example, the extension of a program that computes a function is that function as a mathematical object. Contrast to *intension*.

Filter: A filter is a means to control *polyvariant* analyses, e.g., *binding-time analysis*. A filter is an expression that maps some abstract properties (e.g., binding times) of the arguments of a function to an index value. The analysis combines arguments with identical index and hence determines at most as many different results for the function as there are different index values.

Filtering: In literature about *specializing* logic programs, filtering is used as a synonym for *arity raising*. Not related to *filter* above.

Flow: Used both for control flow (which is the set of possible paths through a program during execution) and for data flow (which is a linking of definitions of values to places where these (directly or indirectly) are used).

Flow analysis: An analysis that computes or estimates the flow of a program. The flow may be the desired end result or it may be a means to compute some other desired properties (e.g., *binding-times*).

Flow-sensitive analysis: An analysis that can give different results for different *program points* instead of a single result that applies to all program points. Mostly used in imperative languages where the same variable may get different (abstract) values at different assignment statements.

Folding: The inverse of *unfolding*, whereby an expression is replaced by a call to a function that has this expression as its body. This function may be defined previously or a new function may be created for this purpose.

In *supercompilation*, the word is often used for the related process of replacing a *configuration* by a reference to a previously encountered identical configuration. Folding of this kind can create cycles in the graph of configurations and hence recursive definitions in the transformed program.

Fold-unfold transformation: See *unfold-fold transformation*.

Functional logic languages: Languages that integrate functional and logic programming within a single framework. Functional logic languages with a complete operational principle are based on *narrowing*. Lazy, efficient, functional computations are combined with the expressivity of logic variables, which allows for *function inversion* as well as logical search.

Full laziness: A transformation performed by some compilers for lazy functional languages. In a curried function, subexpressions that only depend on the first arguments(s) are extracted and evaluated only once even if a partial application is used repeatedly with different values for the remaining parameters. This is similar to the loop-hoisting optimization done in optimizing FORTRAN compilers. Full laziness can be used to do a limited form of *data specialization*.

Fusion: Symbolic composition of two functions with the aim of removing the intermediate data structure that holds the result of the inner function before it is consumed by the outer one, by merging the consumer with the producer.

Futamura projections: Equations that show how compilation and compiler generation can be done by *specialization* of interpreters and *self-application* of *partial evaluators*. The first Futamura projection describes how compilation of a program can be done by specializing an interpreter with respect to the program. The second Futamura projection describes how a compiler can be generated by specializing a partial evaluator with respect to an interpreter. The partial evaluator used for specialization is usually identical to the one being specialized, in which case it is an example of self-application. The third Futamura projection shows how a compiler generator can be obtained by specializing a partial evaluator with respect to a partial evaluator. The third Futamura projection too is usually done by (double) self-application of a single partial evaluator. The compiler generator transforms interpreters into compilers. Named after the originator, Yoshihito Futamura.

Gegen: A program that creates a *generating extension* given a subject program and an *initial division*. See also *cogen*.

Generalized restart: (obs.) Refers to the fact that a potential loop has been detected and that *specialization* is restarted with a more general expression.

Generalization: Replacing a *configuration* by a less specific configuration with the aim of enabling *folding*. Generalization is mainly useful when an unbounded number of different configurations are generated, as generalization can map these to a finite number of less specific configurations.

Generating extension: A generating extension of a program p is a program p_{gen} which given an input v returns a version of p that is *specialized* to v, i.e., the *residual program* of p with respect to v. A generating extension of p can be obtained by the *second Futamura projection* (the generating extension of an interpreter is a hence a compiler, but the principle extends to other types of programs). A program that can produce generating extensions from programs can (indirectly) be used to perform *partial evaluation*. Such a generator of generating extensions can be produced by the *third Futamura projection*. The term is due to Andrei Ershov.

Global control: Control devoted to ensuring the closedness of the *residual program* without risking non-termination. This is usually achieved by using an abstraction operator. See also *local control*.

Incremental specialization: Specialization that is done in *stages* either as the *static* data becomes available or by demand as parts of the *residual program* are needed.

Independence: A condition which is needed to ensure the *strong soundness* of *partial evaluation* for (*functional*) *logic programs*. It essentially guarantees that different *specializations* for the same definition are correctly distinguished and is generally achieved by a *post-processing renaming* transformation. In the case of (pure) logic programming, only (pairwise) nonunificability of the partially evaluated calls (atoms) is required for ensuring the independence of the resultant procedures. In functional logic programs, also overlaps among the specialized calls must be forbidden.

Inherited limit: A limitation of a *partial evaluator* which means that no *residual program* produced by this partial evaluator can use a particular language feature (in terms of size, complexity, number etc.) more than the original program. For example, with some partial evaluators, the number of functions in the residual program cannot exceed the number of functions in the original program, or the types in the residual program can be no more complex than those in the original.

Initial division: The user's specification of the *binding times* of the input and output values for a program to be *partially evaluated*. Often used as input for *binding-time analysis*.

Inlining: A synonym for *unfolding* of functions; used mostly when unfolding is not essential for the progress of the program transformation. Often done as a *postprocess* to improve readability or efficiency of residual programs.

Intension: Properties of an object that can be seen from its name or text. For example, the intension of a program that computes a function is the program's textual representation. Contrast to *extension*.

Internal specialization: Specialization of components of a program to exploit local information or context. Internal specialization does not change the functionality of the program as a whole and can be seen as an advanced compiler optimization analogous to *constant folding*.

Interpretive approach: An instance of *metasystem transition*. Indirect *specialization* of a program through an intervening interpreter. The interpretive approach can increase the power of the specializer by maintaining appropriate information in the interpreter.

KMP-test: A classical benchmark for *partial evaluators*. The goal is to obtain an efficient pattern matcher—similar or identical to the well-known Knuth-Morris-Pratt (KMP) algorithm—by *specializing* a naive, inefficient pattern matcher with respect to a given pattern.

Let-insertion: Systematic abstraction of the arguments of functions and primitives using let-expressions, so these are applied to the bound variables instead of to the argument expressions. Let-insertion is an essential means to preserve the semantics when *specializing* functional programs with effects, as it can avoid discarding, duplication, and reordering of effectful computations during specialization. Also used as a *generalization* mechanism in *deforestation* and related transformations. The correctness of let-insertion can be justified by Eugenio Moggi's computational calculi.

Lift: To cast a value with an early *binding time* to a value (constant) with a late binding time. Lifting a value usually requires it to be *reified*, i.e., converted to a printable representation. The origin of the word lies in *binding-time analyses* that were based on a lattice of binding-time descriptions where late *binding times* are "above" early ones. See also *cross-stage persistence*.

Also used, by extension, to mean any conversion between different representations of a value in a *partial evaluator*.

Loader: See *data specialization*.

Local control: Control of the construction of partial computation trees for single calls in an automatic algorithm for *partial deduction*. See also *global control*.

Memoization: The act of saving the *static* state with respect to which a *program point* has been or will be *specialized*. The objective may be to enable *code sharing* or to later restore the state prior to the program point actually being specialized.

The term is also used about an optimization strategy whereby results of function calls are stored with the purpose of avoiding reevaluation if the function is called again with the same arguments.

Meta-: A prefix used to indicate objects or methods that manipulate other objects or methods, e.g., when programs are used as objects that are manipulated by other programs.

Metasystem transition: Moving from a level to its metalevel, e.g., moving from execution of programs to treating these as objects to be manipulated by other programs. Mostly used in literature about *supercompilation*.

Mix: (obs.) A common, especially in older literature, name for a *partial evaluator*, derived from *mixed computation*.

Mix equations: (obs.) See *Futamura projections* and *Partial-Evaluation equations*.

Mixed computation: A process that given a program p and some data d produces a new program p' and some new data d' such that running p' on d' produces the same result as running p on d, i.e., $[\![p]\!]\,d = [\![p']\!]\,d'$. Has often been used synonymously with *partial evaluation* but is more general, since it also includes *data specialization*.

Monogenetic: Having a single origin. With a monogenetic *specializer*, each construct in the specialized program can be attributed to one specific occurrence of the construct in the source program. The converse of *polygenetic*.

Monovariant: Having or creating a single *variant*. Used about analyses or transformations. The converse of *polyvariant*.

Most specific generalization (msg): Used for *generalization*. The basic idea is, given two or more *configurations* represented as terms, to compute the single, most specific term of which both initial configurations are instances. Msg is also sometimes referred to as least general generalization, anti-unification, or the least common anti-instance.

Multi-level language: A language where several levels of *staging* (e.g., *binding times*) have been made explicit, e.g., by *annotations*. See *two-level language*.

Multi-stage language: A language that allows construction of program fragments through *quasi-quotation* and execution of these fragments within the same execution process as the constructing program. A specal case of *Multi-level languages*.

Narrowing: A unification-based, parameter passing mechanism which extends functional evaluation through goal-solving capabilities as in logic programming. Narrowing is complete in the sense of functional programming (computation of normal forms) as well as logic programming (computation of answers). In order to avoid unnecessary computations and to deal with infinite data structures, demand-driven generation of the search space has been recently used in (different kinds of) lazy narrowing strategies. Narrowing is similar to *driving*.

Narrowing-driven partial evaluation (NPE): A generic *partial evaluation* framework for *functional logic languages* based on an automatic, on-line PE algorithm that builds narrowing trees. By virtue of narrowing, the NPE scheme combines the propagation of partial data structures (by means of logical variables and unification) with highly efficient evaluation strategies (thanks to the functional dimension). The NPE framework provides the same potential for *specialization* as *positive supercompilation* of functional programs and *conjunctive partial deduction* of logic programs.

Needed narrowing: Currently the best (lazy) *narrowing* strategy for *functional logic programs* due to its optimality properties with respect to the length of derivations and the

number of computed solutions. Needed narrowing is defined on inductively sequential programs (that are similar to functional programs with case expressions).

Needed narrowing-based partial evaluation (NNPE): A (strongly correct) instance of the NPE scheme which uses *needed narrowing* both for execution and for *specialization*. In contrast to other PE methods for *functional logic programs*, the NNPE process preserves the determinism properties of the original program as well as its (inductively sequential) structure, which is a prerequisite for optimal evaluation strategies. NNPE is able to produce both *polyvariant* as well as *polygenetic* specializations.

Negative driving/supercompilation: "Ordinary" (positive) driving/supercompilation will only propagate positive information, i.e., the fact that a certain variable must match some given expression. Negative driving/supercompilation also propagates the information that certain variables definitely do not match a given expression.

Normalization: Rewriting a term to some kind of normal form. Supplying a program with a part of its input yields a normalizable term; its normalization corresponds to *partial evaluation* with respect to this input.

Normalization by evaluation: Normalization using a "normalization function" instead of repetitive reduction. Related to *type-directed partial evaluation*.

Offline: Offline steps are done in a separate (usually earlier) *stage* from the "main" steps of a process. In terms of compilers, steps done during compilation (e.g., static type checking) are offline with respect to running the compiled program. In terms of *partial evaluation*, offline refers to steps or analyses done on the program to be *specialized* before the static data with respect to which it is specialized is known. A partial evaluator is usually called offline if it relies on an offline *binding-time analysis*. See also *online*.

Online: Online steps are done at the same time as or interleaved with the "main" steps of a process. For example, run-time type checking is online whereas compile-time type checking is offline with respect to running a program. In terms of *partial evaluation*, online refers to steps performed during partial evaluation, i.e., when the *static* data are known. A partial evaluator is called online if it doesn't use a priori knowledge (based on *binding-time analysis*) of which values are static but checks for this at every operation done on the data during partial evaluation. Since binding-time analysis is approximative, online partial evaluators may be able to exploit more static data than offline partial evaluators. See also *offline*.

Optimal partial evaluator: A partial evaluator that is capable of completely removing the overhead of self-interpretation when compiling a program by *specializing* a *self-interpreter* with respect to the program.

Partial completion: A completion-based technique for the synthesis of functional programs which is closer to *fold/unfold transformations* than to *partial evaluation*, as it requires a heuristic for creating "eureka" rules.

Partial deduction: Synonym for *partial evaluation* of pure logic programs.

Partial evaluation: Partial evaluation is similar to normal evaluation but has incomplete information, so some parts of the program must be left unevaluated. The unevaluated parts are combined to produce a *residual program*, which when executed with the remaining information produces the same result as if the original program were executed with complete information. The aim is that the residual program will use fewer resources (time and/or space) to compute the result. Transformations such as *supercompilation* or

deforestation that rely on the construction and decomposition of intermediate terms are not normally considered partial evaluation, though their effects can be similar. Partial evaluation is often used for *specializing* a program with respect to incomplete input, usually in the form of values of some of several parameters. Another application is *internal specialization*.

Partial evaluator: A program capable of performing *partial evaluation*.

Partially static: A compound value with both *static* and *dynamic* components, and by extension variables and expressions that have partially-static values.

PE: Abbreviation for *partial evaluation*.

Pecom: See *cogen*.

Partial-evaluation equations: Equations that relate *partial evaluation* of a program to obtain a *residual program* and execution of the residual program to execution of the original program. Also called *Mix equations*. See *Futamura projections*.

Phase distinction: (obs.) The separation of different *binding times* or *stages of computation*. Originally used in type theory.

Polygenetic: Having multiple origins. With a polygenetic specializer, a construct in the *specialized* program may be attributed to one or more occurrences of the construct in the source program. For example, a program in the residual program my be the composition of several functions from the original program. The converse of *monogenetic*.

Polyvariant: Having or creating several *variants*. Used about analyses or transformations. The variants are usually created by a variable or parameter taking on several different abstract or concrete values (e.g., binding times or natural numbers) at different times, causing a variant for each value to be created. The converse of *monovariant*.

The phrasing of this entry is necessarily vague because it is used for a variety of similar notions. See the following entries for clarifications of specific instances.

Polyvariant analysis: Usually, a program analysis that, depending on different contexts, can compute several different abstract properties for a single *program point*. For example, a *binding-time analysis* that allows the same function to be called with several different *binding-time patterns* instead of merging these into a single most general (dynamic) pattern.

Polyvariant program-point specialization: A specialization technique that can generate several differently named specialized *variants* of the same procedure or *program point*, allowing them to refer to each other by their names. This differs from what can be obtained by *unfolding* alone since polyvariant program-point specialization additionally requires *suspension* or *folding*.

Polyvariant specialization: A specialization technique that can generate several different specialized *variants* of the same procedure or *program point*, e.g., by *unfolding* or *program-point specialization*.

Positive driving/supercompilation: See *negative driving/supercompilation*.

Postprocessing: Any steps of the *partial evaluation* that are performed after a tentative form of the *residual program* has been produced. Common postprocessing steps include *arity raising*, *function inlining* and reintroduction of various kinds of syntactic sugar.

Preprocessing: When used in relation to *partial evaluation*, preprocessing is any analysis or transformation of the program that is performed *offline*, i.e., before *specialization* as such begins.

Program point: An occurrence of an expression or statement in a program. Program points are typically units of program that are important to control-flow in the program, such as basic blocks, function definitions or function calls. See also *polyvariant program-point specialization*.

Projection: Sometimes used as a synonym for *partial evaluation*, since *specializing* a function with respect to fixed values of some parameters is similar to projecting a function to a sub-space of its domain. Also used in the domain-theoretic sense for an idempotent function that is smaller (in the domain ordering) than the identity function. Domain-theoretic projections have been used for *binding-time analysis*.

Quasi-quotation: Marking parts of a program to indicate that these parts are not to be evaluated but should instead be used to construct a piece of program text which may be written out or executed later. Quasi-quotation is often combined with an un-quotation mechanism that allow computed values to be inserted in the constructed program text.

Reader: See *data specialization*.

Reflection: Turning a textual object into an executable object or function, i.e., going from *intension* to *extension*. For example, transforming the text of a function into the function itself. Mostly used in combination with *reification*.

Reification: Turning an executable object or function into a textual object, i.e., going from *extension* to *intension*. For example, transforming a function into its textual definition. Mostly used in combination with *reflection*. See also, *lifting*, *metasystem transition* and *type-directed partial evaluation*.

Renaming: Renaming is used within *polyvariant specialization*, and gives each variant a different name to avoid clashes. Is often used in conjunction with *arity raising* and *filtering*.

Residual: Belonging to the *residual program*. Used, e.g., for 'residual types', 'residual functions', etc..

Residualize: Transforming or adding *annotations* to a program to force a selected value or action to become *dynamic*. This is typically done to avoid *code explosion* or non-termination.

Residual program: A program obtained by specializing a program with respect to some of its arguments. See *partial evaluation*.

Return sensitive: Used for analyses that can give different results for the return value of a function and the body (i.e., side-effects) of the function, instead of a single result to describe both.

Run-time code generation: A process by which a program generates code as it runs for the purpose of executing the code later during the same run. In general, generation and execution of code can be arbitrarily interleaved in a single run.

Run-time specialization: A form of *run-time code generation* where the generated code is obtained by *partial evaluation* with respect to data that is obtained at run time.

Self-interpreter: An interpreter whose defined language is a superset of its defining language. Such an interpreter is *self-applicable* and thus it can run itself.

Self-applicable: A program or function capable, in a useful way, of being applied to itself or its own text. A *partial evaluator* is called self-applicable if non-trivial instances of the second or third *Futamura projections* exist.

Self-application: Applying a function or program to itself. See *self-applicable*.

Sensitive: Used about analyses that take certain aspects into account, i.e., is sensitive to these aspects. See *flow sensitive*, *context sensitive* and *return sensitive*.

Slicing: A transformation that extracts parts of a program that share a common property. These parts are called a "slice". Usually, the slice is required to be an executable program. Two common cases are forward slices (the parts of a program that may depend on a particular variable or expression) and backwards slices (the parts of a program thet may influence a particular variable or expression). Backwards slicing has been suggested as a means to achieve *specialization*.

S_n^m **theorem:** Theorem by Kleene giving a key closure property for (partial) recursive functions. In terms of programs (that represent functions) it goes as follows: If S is a program with $m+n$ parameters and $a_1, \ldots, a_m$ are values, then there exists a program S' such that for any set of values $b_1, \ldots, b_n$, running S' on $b_1, \ldots, b_n$ gives the same result as running S on $a_1, \ldots, a_m, b_1, \ldots, b_n$. Furthermore, S' can be found algorithmically from S and $a_1, \cdots, a_m$. The S_n^m theorem is the theoretical basis for *specialization*.

Specialization: Restricting a program to be used in a subset of the ways it was originally intended. The restriction can be specified in a variety of fashions, such as providing part of the input, part of the output, or properties to be satisfied by the input or the output. The term normally suggests that the specialized program should exploit the given restriction to gain efficiency when computing in the restricted subset (see also *trivial specialization*). The specialized program can be realized in a variety of fashions but specialization is often used as a synonym for *partial evaluation* which is actually just one of many techniques for specialization. Other techniques include *supercompilation*, *data specialization* and *slicing*.

Specialization class: An extension to an object system that provides a declarative mechanism for specifying invariants that can be exploited by a *specializer*.

Specialization point: A *program point* that will be specialized to named *residual* program points, which will be shared if the specialization point is repeatedly specialized in the same *static* state.

Specializer:

1. A software system that performs *specialization*.
2. The part of a *partial evaluator* that has knowledge of the concrete values of the *static* data and uses this to transform the program. The specializer may be supplemented by *preprocessing* (e.g., *binding-time analysis*) and *postprocessing* (e.g., *arity raising*).

Specializer projections: A generalization of the *Futamura projections*. Equations that show how *partial evaluation* may achieve specialization and generation of specializers using *self-application*.

Stage of computation: A part of a computation that only depends on values computed by itself or on values already available. Each stage of a computation may have a separate *binding time*.

Staging transformation: Program transformations that shift computations between *stages*, e.g., loop hoisting, strength reduction, *syntactic currying* or *partial evaluation*. The aim of the reordering of computations is to reduce the total cost of computation or change how it is distributed over time.

Static: Having an early *binding time*.

Strong correctness: *Soundness* and *completeness* of a transformation with respect to computed answers (and values). It is based on two main properties: *closedness* and *independence*.

Soundness: The transformed (specialized) program does not produce, for the intended queries, any additional answer (or value) which is not produced by the original program.

Structure splitting: Replacing, in a *residual program*, a single structured value by several distinct values. Each new value holds a portion of the original structure. For example, if the original program has a variable that contains a list of values, this may in the residual program be replaced by a number of variables, each holding one element of the list. Parts of the structure that are not used in the residual program may be eliminated. This may in some cases cause the entire structure to be eliminated.

Subject program: The program text that is given as input to a program transformation (e.g., partial evaluation).

Supercompilation: Short for "Supervised Compilation". A program-transformation technique that combines *unfolding*, *folding*, *driving* and *generalization*. Can be used for *partial evaluation*, *fusion*, *tupling* and other transformations. Due to Valentin Turchin.

Suspension: When using *unfolding* and *folding*, a term that is not unfolded or folded can be suspended. A suspended call is replaced by a reference to a new definition which must be produced later. Also used as a synonym for *residualization* and for the completely unrelated concept of a "thunk" (or parameterless closure) that delays the computation of an expression.

Symbolic evaluation/execution: Evaluation of a term or execution of a program by using algebraic laws instead of or in addition to arithmetic (or normal evaluation/execution).

Syntactic currying: Currying being applied as a program transformation as opposed to semantic currying done at run-time by applying a currying combinator.

Template: Code constructed off-line for a piece of code belonging to a late *stage of a computation*. Contains "holes" to be filled in with the values computed by an earlier stage. Specialization using templates is done by pasting together the templates guided by the computation of an earlier stage and filling the holes. Templates are used, e.g., in *run-time code generation*.

"The Trick": A *binding-time improvement* where a use of a *dynamic* value is replaced by a dynamic case analysis where each case is a use of a *static* value. The static context is duplicated to each static value, so it will be *specialized* for each of these. The Trick requires static knowledge of the range of the dynamic value, so that the possible cases can be enumerated during specialization. Closely related to *eta-expansion* for sums.

Trivial specialization: Specialization by prepending a specialized function that simply calls the original function with constant values for the *static* parameters. Trivial specialization is an instance of the S_n^m *theorem*, and achieves no speedup.

Tupling: Program transformation where two (or more) functions with overlapping input are combined to a single function in order to avoid multiple traversals of the shared input.

Two-level language: A language where two levels of *staging* (e.g., *binding times*) have been made explicit, e.g., by *annotations*.

Type-directed partial evaluation: A partial-evaluation technique where the type of the *residual program* and an executable version of the original program applied to the *static*

arguments are used to produce a residual program without *symbolically executing* the original program (but instead executing it natively in a non-standard context). Type-directed partial evaluation uses a combination of *reflection*, *reification* and *eta-expansion* akin to *normalization by evaluation*.

Type specialization: A *partial-evaluation* technique where a program and its type are specialized to a new program with a new type. Typically, type specialization can produce *residual* types that are arbitrarily more complex than the original types and hence has no *inherited limit* on the type structure.

Unfold-fold transformation: A program transformation framework for recursive equations which is general enough to describe many forms of *partial evaluation* and program *specialization*. It is based on five ways of transforming sets of definitions: *unfolding*, *folding*, introduction of new definitions, instantiation of existing equations, and use of laws about primitive operations.

Unfolding: Replacing a name by the structure it denotes, e.g., replacing a function name by its definition. Unfolding is often combined with instantiation or local bindings to create a new instance of the structure.

Use-sensitive: Used for analyses that can give several results for a single *program point* with the intent of using the different results in different contexts. For example, if the result of an expression may be used both *statically* and *dynamically*, both a static and a dynamic value are computed and the appropriate one selected at each use.

Variable splitting: Usually, an instance of *structure splitting* but as occasionally been used for *binding-time separation*.

Variant: One of several objects, e.g., functions or *annotations*, related by having a common origin, e.g., by being different *specializations* of the same original function or different annotations of the same *program point*.

Weakening: Program transformation induced by logical weakening of the postcondition and logical strengthening of the precondition.

Whistle: A function that checks whether the current *configuration* must be *generalized* with respect to a preceding configuration. Used to avoid nontermination in *supercompilation*. The most successful whistle uses homeomorphic embeddings.

References

1. Torben, Æ. Mogensen and Carsten Kehler Holst. Terminology. In *Partial Evaluation and Mixed Computation*, Dines Bjørner, Andrei P. Ershov, and Neil D. Jones (Eds.). North-Holland, 1988, pp. 583–587.